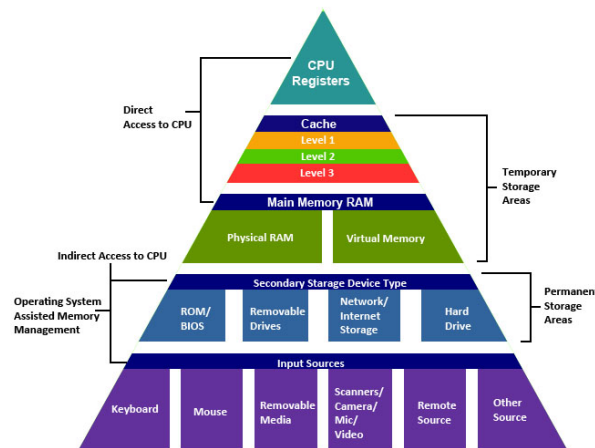


## אחזור מידע

### Index Construction

הנושא המרכזי של הרצאה זו הוא תהליך בניית האינדקסים (ה-Indexing / אינדוקס) וכיצד הוא נעשה. אנו דיברנו על תהליך בניית האינדקסים בהרצאה הראשונה, אולם ההנחה שלנו הייתה שכמות המסמכים שלנו היא כזו שהן המילון שאנו יוצרים והן ה-posting lists הם קטנים דיים, כך שהם יכולים להישמר הזיכרון המחשב בשלמותם. כזכור, כחלק מפעולת האינדוקס, אנו צריכים למיין את ה-terms השונים, עד כה הנחנו שאין לנו בעיה, ואנו יכולים לבצע פעולה זו בשלמותה בזיכרון המחשב (ה-RAM). בהרצאה זו אנו נהייה יותר מציאותיים. אנו נזנח את ההנחה שלנו לגבי גודלם של המסמכים שלנו (הקורפוס), ונראה באלו אסטרטגיות אנו יכולים להשתמש כאשר הזיכרון הזמין לנו מוגבל ואינו מאפשר את ביצוע הפעולות השונות בזיכרון המחשב (מפאת כמות הנתונים וגודלם שאינם יכולים להישמר בשלמותם בזיכרון).

לפני שנדבר על האלגוריתמים עצמם אנו נתחיל עם סקירה בסיסית של חומרת המחשב, היות ובתחום אחזור נתונים הרבה החלטות הנוגעות לעיצוב המערכת מבוססות על מבנה החומרה שלנו. העיקרון הראשון שאלינו אנו צריכים להתייחס הוא שגישה לנתונים הנמצאים בזיכרון מהירה **פי כמה** המאשר לנתונים המאוכסנים על גבי הדיסק הקשיח.



אחד מהגורמים לזמני הגישה הגבוהים לנתונים המאוכסנים על גבי הדיסק הקשיח הוא מה שנקרא Seek Time, שמורכב משלושה גורמים: (1) הזמן שלוקח למקם את הראשים של הדיסק הקשיח במקומם, (2) הזמן שלוקח לשוב את הדיסק ולהביא את הנקודה הנכונה אל מתחת לראשים, ו-(3) זמן הקריאה עצמו של הראשים (אנו נתייחס לשתי הפעולות הראשונות ב-Seek Time). מכוון שב-Seek Time, לא מתבצעת העברת נתונים, עדיף להעביר של גוש אחד גדול של נתונים, שהינה פעולה מהירה יותר, מאשר להעביר של מספר גושים קטנים של נתונים.

נקודה נוספת שדורשת התייחסות היא שפעולות I/O של הדיסק נעשות בגושים, הנקראים בלוקים. כלומר כתיבה או קריאה של נתונים לא נעשית ברמת הבית, אלה בבלוקים (גוש נתונים), המכילים מס' בתים כל אחד. מכוון שברוב המקרים, לאחר שאנו קוראים בית אחד, אנו מעוניינים בקריאה של הבית שנמצא אחריו בקובץ, קריאה של בלוק שלם תהייה יעילה יותר מבחינת זמני התגובה. גודלו של בלוק ממוצע הוא בין 8KB ל-256KB.

בשרתי IR ממוצעים, הזיכרון הוא של מספר GB, לפעמים עשרות. גודלו של הדיסק הקשיח הוא פי כמה (2-3) גדול מהזיכרון הקיים במחשב.

נקודה נוספת אליה נתייחס היא מניעת תקלות בשרתי IR. מניעת תקלות מאד יקרה, ולכן במקום מחשב אחד עמיד ויקר עדיף כמה (עשרות/מאות/אלפים) מחשבים פשוטים וזולים.

הטבלה הבאה מציגה מספר ערכים הקשורים לחומרה אשר ישמשו אותנו בהמשך בדוגמאות השונות שלנו. יש לקחת בחשבון שהערכים הללו הם לא ערכים מדויקים, ובהחלט יתכן שהם השתנו (והתקצרו) במהלך השנים. אולם, הם עדיין מהווים מדד לסדרי הגודל השונים בעבודה בזיכרון המחשבים ובעזרת הדיסק הקשיח.

ערך	משמעות	סימון
5 ms = $5 \times 10^{-3}$ s	average seek time	s
0.02 $\mu$ s = $2 \times 10^{-8}$ s	transfer time per byte	b
$10^9$ s <sup>-1</sup>	processor's clock rate	
0.01 $\mu$ s = $10^{-8}$ s	low-level operation (e.g., compare & swap a word)	
several GB	size of main memory	
1 TB or more	size of disk space	

נקודה נוספת שאליה אנו צריכים להתייחס היא אוסף היצירות של שייקספיר, שאינו גדול מידי בכדי להדגים את הסוגיות בהן עוסקת הרצאה זו. ולכן, לצורך ההדגמה אנו נשתמש באוסף מסמכים אחר, Reuters RCV1 collection, אף שגם הוא לא נחשב גדול, הוא מספיק לצורכי ההרצאה שלנו. אוסף מסמכים זה הוא אוסף של ידיעות חדשותיות שהופצו ע"י Reuters למערכות עיתונים שונות, בין השנים 1995-1996, במהלך 12 חודשים.



You are here: Home > News > Science > Article

Go to a Section: U.S. International Business Markets Politics Entertainment Technology Sports Oddly Enough

### Extreme conditions create rare Antarctic clouds

Tue Aug 1, 2006 3:20am ET

[Email This Article](#) | [Print This Article](#) | [Reprints](#)



SYDNEY (Reuters) - Rare, mother-of-pearl colored clouds caused by extreme weather conditions above Antarctica are a possible indication of global warming, Australian scientists said on Tuesday.

Known as nacreous clouds, the spectacular formations showing delicate wisps of colors were photographed in the sky over an Australian meteorological base at Mawson Station on July 25.

A Reuters RCV1 document

הטבלה הבאה מתארת מס' נתונים סטטיסטיים לגבי אוסף המסמכים RCV1.

ערך	משמעות	סימון
800,000	documents	N
200	avg. # tokens per doc	L
400,000	terms (= word types)	M
6	avg. # bytes per token (incl. spaces/punct.)	
4.5	avg. # bytes per token (without spaces/punct.)	
7.5	avg. # bytes per term	
100,000,000	non-positional postings	

מדוע יש הבדל בין מס' הבתים הממוצע של token ומספר הבתים הממוצע של term? הסיבה לכך היא ש-term הוא ייצוג בודד של מס' tokens זהים. מכוון שיש הרבה tokens קצרים (stop words בעיקר), הרי שהממוצע של אורכי ה-term tokens היה הנמוך מהממוצע של אורכי ה-term. תכונה זו הינה תלויה בשפה, במקרה זה, אנגלית.

## מיון

כזכור מההרצאה הראשונה, בתהליך בניית האינדקס או עוברים על כל המסמכים, אחד אחרי השני. כל מסמך עובר תהליך של tokenization שלאחריו הוא עובר דרך ה-linguistic module, שבעקבותיו או מקבלים רשימה של terms הנמצאים במסמך. בסוף התהליך, הרשימה המלאה של ה-term (מכל המסמכים ביחד), ממוינת תחילה בסדר לקסיקוגרפי, ואח"כ, כמיון משני, לפי ה-docID. את תהליך המיון הנ"ל כולו ביצענו בזיכרון המחשב (ב-RAM).

אם נניח שכל term, לפני בניית רשימת ה-term עצמה, תופס 12 בתים, עבור רשימה גדולה מאד של מסמכים או מדברים על כמות עצומה של זיכרון. במקרה של RCV1 או מדברים על  $T = 100,000,000$ . מכוון שאת הרשימה המלאה של ה-term למיון או יכולים לקבל רק בסוף תהליך הבנייה, הרי שבמקרה הזה המשמעות היא שאנו צריכים 1.12GB של זיכרון לצורך שמירת ה-term עצמם. במחשבים של היום, במקרה הזה, עדיין ניתן לבצע את המיון בזיכרון.

למרות גודלו, RCV1 אינו נחשב לאוסף גדול של מסמכים. למשל, ה-New-York Time, מאפשר לחפש במסמכים שנכתבו במשך יותר מ-150 שנה. ולכן, או צריכים אלגוריתמים למיון, שעובדים בשלבים, ויכולים לשמור תוצאות ביניים על הדיסק.

האם ניתן להשתמש באותם אלגוריתמים למיון (שעובדים בזיכרון), אולם במקום להשתמש בזיכרון המחשב, נשתמש בדיסק? נניח שיש לנו 100,000,000 רשומות שיושבות על הדיסק. נניח שלצורך פעולת השוואה או צריכים שתי פעולות seek, שהן, כאמור, פעולות איטיות מאד (5ms) בהשוואה לקריאה מהזיכרון. וכן, על מנת למיין N רשומות יש צורך ב- $N \log N$  השוואות, אזי במקרה שלנו או מדברים על  $2 \times 100,000,000 \log 100,000,000 \times 5 = 8,000,000,000$  ms, שזה 92.6 ימים. ולכן, התשובה היא לא. או לא יכולים להשתמש באלגוריתמים הרגילים למיון כאשר המידע יושב על הדיסק הקשיח.

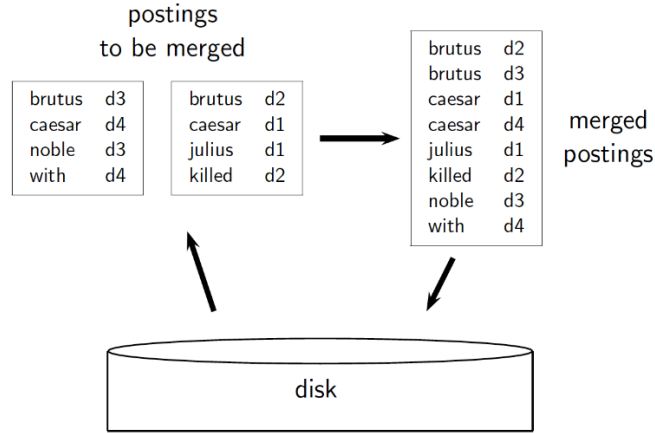
## BSBI: Blocked sort-based Indexing

כאמור, יש לנו 100M רשומות בגודל של 12 בתים (term, docID, freq) שהתקבלו מעיבוד מסמכים. או צריכים למיין את הרשומות לפי השדה term, ומיון משני לפי השדה docID. מכוון שלא ניתן (מכוון שיש מגבלות זיכרון) לשמור את כל ה-100M רשומות בזיכרון המחשב (לצורך המיון), או צריכים למצוא דרך אחרת לבצע את המיון. לצורך כך, נחלק את ה-100M הרשומות שלנו לבלוקים קטנים יוצר, נניח בלוקים בגודל של  $\sim 10M$ , באופן כזה שניתן לשמור שני בלוקים בזיכרון המחשב.

אופן בניית הבלוקים הוא פשוט. או נעבור על קבצים שלנו, ועבור כל term שנקבל, נרשום אותו ואת ה-docID שלו לבלוק הנוכחי. או נבצע תהליך זה עד שהבלוק שלנו יתמלא, אז נשמור את הבלוק על הדיסק. נמשיך לעבור על הקבצים שלנו, כאשר או מתחילים למלא בלוק חדש. כך נעשה עד שנסיים לעבור על כל הקבצים.

כעת, או נמיין כל אחד מהבלוקים שלנו (בנפרד). את זה ניתן לבצע בזיכרון, מכוון שהבלוקים הינם קטנים יחסית. לאחר מכן, נאחד את כל הבלוקים (ששמורים על הדיסק) לרשימה ממוינת אחת.

את תאריך האיחוד, ניתן לעשות באופן זה לאופן שבו איחדנו, או סרקנו רשומות קודם לכן. או נטען לזיכרון את ההתחלה של הבלוק הראשון ואת ההתחלה של הבלוק השני. או נתחיל לעבור על שתי הרשימות, ובצע תהליך הדומה ל-merge sort. כאשר נגיע לסוף של המידע שנמצא בזיכרון של אחד מהבלוקים, נטען חלק נוסף לזיכרון.



להלן הפסאודו-קוד של האלגוריתם הנ"ל. בשורה 1,  $n$  הוא מס' הבלוק, שמאותחל לערך 0. כל עוד לא סיימנו לעבור על כל המסמכים (שורה 2), אנו מגדילים את ערכו של  $n$  ב-1 (שורה 3), ממלאים את הבלוק הנוכחי ברשימה של הזוגות (term, docID) (שורה 4), ממיינים אותו (שורה 5) ושומרים אותו לדיסק (שורה 6). לאחר שעברנו על כל המסמכים, אנו ממוזגים את הבלוקים הממוינים, ששמורים על הדיסק, לרשימה אחת גדולה וממוינת (שורה 7).

**BSBINDEXCONSTRUCTION()**

```

1   $n \leftarrow 0$ 
2  while (all documents have not been processed)
3  do  $n \leftarrow n + 1$ 
4      $block \leftarrow \text{PARSENEXTBLOCK}()$ 
5      $\text{BSBI-INVERT}(block)$ 
6      $\text{WRITEBLOCKTODISK}(block, f_n)$ 
7   $\text{MERGEBLOCKS}(f_1, \dots, f_n; f_{\text{merged}})$ 

```

עד עכשיו עסקנו במיון רשימות ה-(term, docID), כאשר השלב הבא הוא מעבר על הרשימה הממוינת הזאת, בניית המילון וה-posting lists. אנו יכולים לעבוד בצורה זו כל עוד שניתן לשמור את המילון בזיכרון. למעשה, בכדי לחסוך בזיכרון במקום להשתמש ב-term אנו יכולים להשתמש ב-termID. במקרה כזה, אנו צריכים את המילון (שגדל באופן דינאמי ככל שמספר ה-terms גדל כתוצאה ממעבר על מסמכים נוספים) בכדי לבצע את המיפוי של הביטויים (term) למס' המזהה שלו (termID). אמנם, ניתן להשתמש במיקומים מבוססים על הביטוי ומזהה המסמך (term, docID) במקום להשתמש ב-termID ו-docID, אולם אז אנו נצטרך לשמור קבצי ביניים גדולים.

**SPIMI: Single-pass in-memory indexing**

כדי לפתור את בעיית הזיכרון של המילון, אנו יכולים להשתמש באלגוריתם SPIMI.

האלגוריתם דומה לאלגוריתם הקודם, עם מס' שינויים. (1) לכל בלוק ניצור "מילון" משלו שהמבנה שלו הוא (term, docID). כך אנו **לא צריכים** לשמור מיפוי של ביטוי למס' ביטוי (term-termID) בין הבלוקים. (2) לא נבצע מיון. נשמור את המיקומים של ה-terms ב"מילון", בהתאם לסדר שהם מתקבלים, בזמן המעבר על הקבצים. (3) בהתאם לשתי הפעולות האלו, אנו יכולים ליצור inverted index עבור כל בלוק (למעשה ניתן לבצע את שלבים 1 ו-2 במקביל). (4) בהמשך, ניתן למוזג את האינדקסים הללו לאינדקס אחד גדול.

להלן הפסאודו-קוד של האלגוריתם SPIMI-Invert.

```

SPIMI-INVERT(token_stream)
1  output_file = NEWFILE()
2  dictionary = NEWHASH()
3  while (free memory available)
4  do token ← next(token_stream)
5     if term(token) ∉ dictionary
6         then postings_list = ADDTODICTIONARY(dictionary, term(token))
7         else postings_list = GETPOSTINGSLIST(dictionary, term(token))
8     if full(postings_list)
9         then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
10    ADDTODICTIONARY(postings_list, docID(token))
11    sorted_terms ← SORTTERMS(dictionary)
12    WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13    return output_file

```

*output\_file* (שורה ראשונה) הוא הקובץ שבו ישמר כל ה-inverted index. אנו מתחילים עם "מילון" ריק (שורה 2), שבמקרה זה ממומש ע"י hash table. כל עוד יש לנו זיכרון פנוי (שורה 3), אנו מבצעים את השלבים הבאים: (1) נקרא את ה-token הבא מהקובץ (שורה 4). למעשה אנו מקבלים את הצרוף (term, docID). (2) אם ה-term לא נמצא במילון (שורה 5), אז ניצור posting list חדש, שיכיל את ה-docID הנוכחי (שורה 6). (3) אחרת, נקרא את ה-posting list של ה-term הקיים, ונוסיף לסופו את ה-docID הנוכחי (שורה 7). (4) במידה ורלוונטי, שורות 8 ו-9 מטפלות במקרים שבהם ה-posting list הוא מלא (למשל אם השתמשנו במערכים). (5) בשורה 10 אנו מעדכנים במילון את ה-posting list של ה-term הנוכחי (הוספה של חדש או עדכון posting list קיים). אחרי שלב זה, יש לנו inverted index עבור הבלוק הנ"ל. אנו צריכים למיין את ה-inverted index הזה (שורה 11), ואז לכתוב אותו לדיסק (שורה 12).

אנו מבצעים את התהליך הזה לכל המסמכים. במהלך התהליך נוצרים מספר קבצים, אותם אנו צריכים למזג בסוף התהליך.

ניתן ליעל את SPIMI ע"י שימוש בכיווץ. כאשר, (1) אפשר לכווץ את ה-terms, ו-(2) אפשר לכווץ את ה-posting lists. נדבר על כך בהרצאה הבאה.

### אינדקסים מבוזרים

בדוגמאות הקודמות פתרנו את בעיית הזיכרון – כיצד ניתן לבנות את ה-inverted index כאשר הזיכרון שלנו מוגבל. עשינו זאת ע"י ניצול הדיסק הקשיח. אולם במקרים מסוימים (למשל, כאשר אנו רוצים לבנות אינדקס של כל אתרי האינטרנט בעולם), גם גודלו של הדיסק הקשיח מהווה מגבלה.

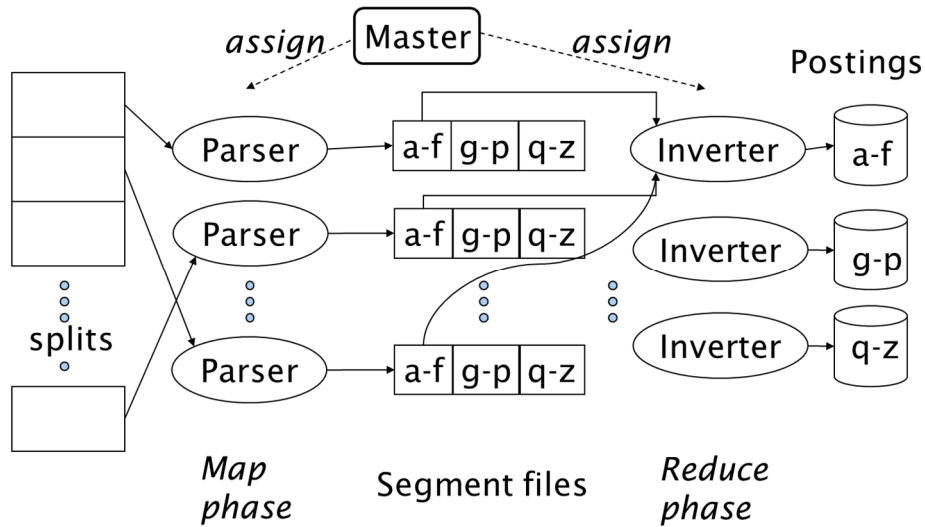
נניח שרוצים לבצע אינדקס של דפים באינטרנט, מהסיבות שהזכרנו, אנו לא נשתמש במחשב בודד, אלא נשתמש באשכול מחשבים מבוזר. מעבר לכך, מחשבים יכולים באופן בלתי צפוי להתחיל ולעבוד לאט או להתקלקל. אנו נראה כיצד אשכול מחשבים מבוזר יכול לעזור לנו במקרים כאלו.

מרכזי הנתונים של מנועי חיפוש באינטרנט (כמו Google), אתרים שונים בהם מתבצעים תהליכי האינדקס וחיפוש, מכילים מחשבים רבים. בנוסף, הם מפוזרים במקומות שונים ברחבי העולם. ההערכות (נכון ל-2007) הן שלגוגל יש בערך מיליון שרתים, ובהם, בסה"כ, כ-3 מיליון מעבדים/ליבות, מפוזרים במקומות שונים ברחבי העולם.

נניח שיש לנו אשכול עם 1000 מחשבים. לכל מחשב יש uptime של 99.9% (כלומר, כל מחשב עובד ב-99.9% אחוז מהזמן). מהו ה-uptime של המערכת כולה (כלומר, מהו אחוז הזמן שבו כל המחשבים עובדים ביחד) ?  $99.9^{1000} = ?$  36%. זהו אחוז נמוך, אם אנו דורשים שכל המחשבים יעבדו בו זמנית ביחד. למזלנו, אנו לא צריכים שכל המחשבים יעבדו בו זמנית.

נניח שיש לנו אשכול מחשבים שאנו רוצים לבצע את תהליך האינדקס. לשם כך, נקצה מחשב אחד מאשכול המחשבים, מחשב ה"מאסטר", לניהול תהליך האינדקס. כמו כן, נפרק את תהליך האינדקס לקבוצה של משימות, אותן נבצע במקביל. מחשב ה"מאסטר" יקצה את כל אחת מהמשימות למחשב פנוי באשכול המחשבים שלנו. במידה ותתרחש תקלה באחד המחשבים, מחשב ה"מאסטר" יכול להקצות את אותה המשימה (שהוקצתה למחשב התקול) למחשב אחר.

לצורך תהליך האינדוקס אנו נשתמש בשתי קבוצות של משימות מקבילות: (1) Parsers ו-(2) Inverters. נחלק את אוסף המסמכים שלנו לקבוצות (Splits - מקבילים לבלוקים באלגוריתמים BSBI/SPIMI). ה"מאסטר" מקצה כל קבוצת מסמכים למחשב Parser פנוי. ה-Parser קורא את המסמכים ומייצר זוגות של term-docID. ה-Parser כותב את הזוגות הללו ב-j מחיצות שונות, כשכל מחיצה מכילה terms בטווח אותיות שונה. למשל, a-f, g-p, q-z, כשבמקרה זה  $j=3$ . השלב הבא הוא השלמת האינדוקס, והוא נעשה ע"י ה-Inverters. תפקידו של ה-Inverter הוא לקחת את הזוגות של term-docID ממחיצה אחת בודדת, למיין אותם וליצור postings lists אותם הוא רושם לדיסק.



ההחלטה איזה מחשב יהיה Parser ואיזה יהיה Inverter היא באחריות מחשב ה"מאסטר", והיא אינה קבועה. כלומר, ה"מאסטר" יכול להחליף את התפקיד של מחשב מסוים בהתאם לאילוצים הקיימים במערכת.

**MapReduce**

האלגוריתם עליו דיברנו כרגע הוא גרסה של MapReduce. MapReduce (Dean and Ghemawat 2004) הוא framework קונספטואלי רובסטי לעבודה עם מחשבים מבוזרים. סכמטית, MapReduce מתאר שני שלבים: (1)  $map: input \rightarrow list(k, v)$  (2)  $reduce: (k, list(v)) \rightarrow output$

לבניית האינדקסים האלגוריתם פועל באופן הבא: (1)  $map: collection \rightarrow list(termID, docID)$  (2)  $reduce: (termID1, list(docID1), termID2, list(docID2), \dots) \rightarrow (postings\ list1, postings\ list2, \dots)$

לדוגמא:

- ❑ Map:
  - ❑  $d1 : C\ came, C\ c'ed.$
  - ❑  $d2 : C\ died.$
  - ❑  $\rightarrow \langle C, d1 \rangle, \langle came, d1 \rangle, \langle C, d1 \rangle, \langle c'ed, d1 \rangle, \langle C, d2 \rangle, \langle died, d2 \rangle$
- ❑ Reduce:
  - ❑  $(\langle C, (d1, d2, d1) \rangle, \langle died, (d2) \rangle, \langle came, (d1) \rangle, \langle c'ed, (d1) \rangle) \rightarrow (\langle C, (d1:2, d2:1) \rangle, \langle died, (d2:1) \rangle, \langle came, (d1:1) \rangle, \langle c'ed, (d1:1) \rangle)$

### אינדקס דינאמי

עד עכשיו הנחנו שאוסף המסמכים שלנו הוא סטטי. במציאות, אוסף מסמכי קבוע הוא נדיר. מסמכים שונים מתווספים עם הזמן, מסמכים עוברים עדכונים ומסמכים נמחקים. המשמעות היא שהמילון וה-*postings lists* דורשים שינויים ועדכונים. הראשון, צריך לעדכן מיקומים של מושגים שכבר נמצאים במילון, והשני, צריך להוסיף מושגים חדשים למילון.

אחד מהדרכים להתמודד עם בעיה זו היא באופן הבא. אנו נהל אינדקס ראשי גדול, כשמסמכים חדשים יכנסו לאינדקס חדש, קטן. בעת ביצוע חיפוש, הוא יתבצע מול שני האינדקסים והתוצאות יאוחדו לתוצאה אחת. בכדי להתמודד עם מחיקות באופן פשוט, אנו נשמור ווקטור בוליאני (bit-vector) שבו מצוין אם המסמך מחוק או לא (כך שאנו לא באמת צריכים למחוק את המסמך מהאינדקס). כל מסמך שמתקבל מתוצאת החיפוש, נבדק מול הווקטור הנ"ל בכדי לדעת אם הוא נמחק או לא, ואם יש להציג אותו כחלק מהתוצאה או לא.

מידי תקופה, כאשר האינדקס המשני, הקטן, גדל מעבר לגודל מסוים (כך, שלא ניתן לשמור אותו בשלמותו בזיכרון המחשב), אנו בונים את האינדקס הראשי מחדש (כולל את המסמכים הישנים והחדשים).

לשיטה זו מספר חסרונות. (1) בעיות כאשר יש מיזוגים בתדירות גבוהה. (2) ביצועים נמוכים בעת ביצוע פעולת המיזוג. למעשה, ניתן למזג את האינדקס המשני עם האינדקס הראשי בעילות יחסית אם כל *postings list* נשמרת בקובץ אחד. אולם, במקרה כזה, פעולת המיזוג היא למעשה פעולת הוספה (Append), ואנו נזדקק להרבה מאד קבצים - לא יעיל מבחינת מערכת ההפעלה. בהמשך ההרצאה אנו מניחים שיש לנו קובץ אינדקס אחד גדול.

### מיזוג לוגריתמי

מיזוג לוגריתמי הינה שיטה יעילה יותר להתמודדות עם אוספי מסמכים דינאמיים. המיזוג הלוגריתמי עובד באופן הבא. אנו שומרים מספר אינדקסים, כל אחד גדול פי 2 מקודמו. את האינדקס הקטן ביותר ( $Z_0$ ) אנו שומרים בזיכרון. אינדקסים גדולים יותר ( $I_0, I_1, \dots$ ) נשמרים על הדיסק. אם  $Z_0$  גדל מעבר לגודל מסוים,  $n$ , נשמור אותו על הדיסק כ- $I_0$ . אם כבר קיים, אז נמזג אותו עם  $I_0$  כ- $Z_1$ . את  $Z_1$  נשמור כ- $I_1$ , במידה ו- $I_1$  לא קיים, אחרת נמזג עם  $I_1$  בכדי לצור את  $Z_2$ .

```

LMERGEADDTOKEN(indexes, Z0, token)
1  Z0 ← MERGE(Z0, {token})
2  if |Z0| = n
3    then for i ← 0 to ∞
4      do if Ii ∈ indexes
5        then Zi+1 ← MERGE(Ii, Zi)
6           (Zi+1 is a temporary index on disk.)
7           indexes ← indexes - {Ii}
8        else Ii ← Zi (Zi becomes the permanent index Ii.)
9           indexes ← indexes ∪ {Ii}
10         BREAK
11         Z0 ← ∅
    
```

```

LOGARITHMICMERGE()
1  Z0 ← ∅ (Z0 is the in-memory index.)
2  indexes ← ∅
3  while true
4  do LMERGEADDTOKEN(indexes, Z0, GETNEXTTOKEN())
    
```

בפסאודו-קוד, כל פעם שאנו מוסיפים token ל- $Z_0$  (שורה 1), אנו בודקים אם  $Z_0$  הגיע לגודל המקסימאלי שלו (שורה 2). אם כן, אנו מבצעים את המיזוג הלוגריתמי (שורות 3 עד 10). אנו עוברים על כל האינדקסים האפשריים, ובודקים אם הם קיימים (שורות 3 ו-4). אם  $I_i$  קיים, אנו ממזגים את  $I_i$  עם  $Z_i$  ויוצרים את  $Z_{i+1}$  (כאינדקס כזמני על הדיסק), ובמקביל מוחקים את האינדקס  $I_i$  (שורות 4 עד 7). אחרת, אנו יוצרים אינדקס חדש  $I_i$  שהוא למעשה האינדקס  $Z_i$  (שורות 8 ו-9). בסוף התהליך אנו מאפסים את  $Z_0$ .

בהנחה שמספר ה-tokens באוסף המסמכים שלנו (הקורפוס) הוא  $T$ . כאשר אנו עובדים בשיטת האינדקס הראשי והמשני, זמן בניית האינדקס הוא  $O(T^2)$ , מכוון שכל מיקום מטופל בכל מיזוג. כאשר אנו משתמשים במיזוג לוגריתמי, כל מיקום מטופל  $O(\log T)$  פעמים, ולכן הזמן הכולל הוא  $O(T \log T)$ . לפיכך, מיזוג לוגריתמי הוא יעיל יותר לבניית האינדקסים.

מצד שני, כאשר אנו מבצעים שאילתא, אנו צריכים למזג תוצאות של  $O(\log T)$  אינדקסים. מעבר לכך, איסוף סטטיסטיקות במקרה של מיון לוגריתמי קשה יותר. לדוגמא, לצורך תיקון שגיאות, אחת השיטות היא להציג את האלטרנטיבה עם מירב התוצאות. איך משיגים את התוצאה הזאת כאשר יש לנו מספר רב של אינדקסים (אולי להתייחס רק לאינדקס הגדול ביותר).